G.) Specifications of the Fleas Pseudorandom Functions

The Fleas Pseudorandom Functions are the basis of all other cryptographic primitives of the Fleas family. They exist in different variants. Only the most recent variant "develop_ln2" shall be described here. Older variants that are used in Academic Signature can be inspected by looking at the published source code. The module "helpersxx.cpp“ contains all variants.

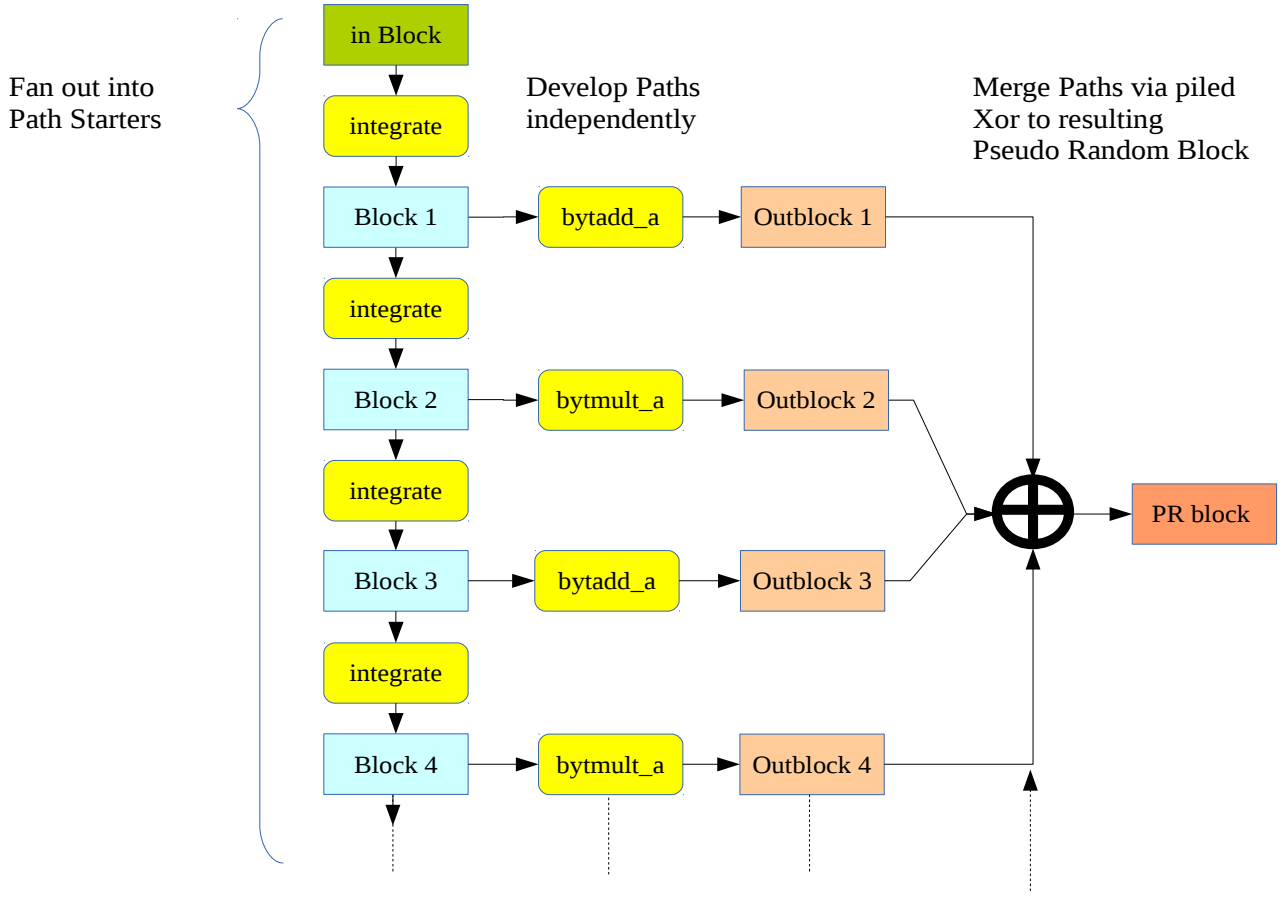
a) Procedure “develop_ln2“

The respective routine in C++ is listed below. It calls three nontrivial subroutines called "bytinteg", "bytadd_a" and "bytmult_a".

```
/************************************************************/
bool develop_ln2( char* ad, int blolen, int rounds, int paths, unsigned char spice, unsigned char
inc)

{
    char *acc;
    char **a;
    bool res;
    unsigned int i,j,max;
    unsigned char offs;



    a=(char **)malloc((paths+1)*sizeof(char *));
    if(a==NULL){throwout(_("allocation error  (0) in developln2"),5); return false;}
    for(i=0;i<(unsigned int)paths+1;i++)
    {
        *(a+i) = (char *) (malloc(blolen));
        if(*(a+i) == NULL)
        {
            for(j=0;j<i;j++)
            {
                free( (void *) *(a+i));
            }
            free( (void *) a);
            throwout(_("allocation error  (1) in developln2"),5);
            return false;
        }
    }
    acc= *(a+paths);
    //all  storage blocks allocated

    offs=spice;  //entry parameter for integration
    memcpy(*a,ad,blolen); //deposit in-block
    res=bytinteg(blolen,*a,offs,inc);
    for(i=1;i<(unsigned int)paths;i++)  //set all integrated in-blocks
    {
        memcpy(*(a+i),*(a+i-1),blolen);
        res=bytinteg(blolen,*(a+i),offs+(unsigned char)i,inc+(unsigned char)i);
    }
    //chained integration blocks set!
    for(i=0;i<(unsigned int)paths;i++)
    {
        if((i&1)==1) //odd
        {
                res= bytadd_a(blolen,*(a+i),*(a+i),rounds,spice,2); //spice is 1, two flea rounds
        }
        else  //even
        {
                res= bytmult_a(blolen,*(a+i),*(a+i),rounds,spice+6,2); //spice is 1, two flea rounds
        }
        if(!res)
        {
            for(j=0;j<i;j++)
```

```
                {
                    free( (void *) *(a+i));
                }
            free( (void *) a);
            throwout(_("bytadd error in develop_ln2"),5);
            return false;
        }
    }
    max=blolen>>2;
    for(j=0;j<max;j++) //do the xor in 4 byte blocks
    {
        *( (ulong32*) ( (acc+(j<<2)))) = 0;
        for(i=0;i<(unsigned int)paths;i++)
        {
            *( (ulong32*) ( (acc+(j<<2)))) ^=  *((ulong32 *) ( (*(a+i)) +(j<<2)));
        }
    }
    for(j=max<<2;j<(unsigned int)blolen;j++) //do the xor for the remaining bytes
    {
        *(acc+j) = 0;
        for(i=0;i<(unsigned int)paths;i++)
        {
                *(acc+j) ^= *((*(a+i)) +j);
        }
    }  //alles gexort - hopefully :-)
    //overwrite inblock
    memcpy(ad,acc,blolen);
    //kill scratchblocks
    for(i=0;i<(unsigned int)paths+1;i++)
    {
        free( (void *) *(a+i));
    }
    free((void *) a);
    return true;
}
/*************************************************************/
```

The figure below shows the structure of the pseudorandom procedure “develop_ln2”.



In a first step the block is developed into n start blocks by successive application of a function that basically performs mathematical integration. Then the start blocks are developed further by application of either the routine bytadd_a for odd path number or bytmult_a for even path number. The resulting path results are then reunited by xoring them all together. Note that the procedures integrate, bytmult_a and bytxor_a each accept additional spice parameters allowing to tweak the output of the procedures. Details on the passed spice parameters are omitted in this diagram for clarity.

The called procedures will now each be specified.

a) Procedure "bytinteg"

The respective routine in C++ is listed below as contained under the name "bytinteg" in the Academic Signature module “helpersxx.cpp“.

```
/*************************************************************/
bool bytinteg(int blolen, char* blo, unsigned char offs, unsigned char inc)
// integriert einmal rauf und einmal runter, add offset am Start und add inc bei jedem Schritt
// dient zum "entcodieren" von Textinfo
// dient auch zur Bitdispersion:Ändere ein Bit am Anfang -> beeinglusse jedes Byte

{
  unsigned int ind;

  if(blolen<2)return false;

  *blo += (char) offs;
  for(ind=1;ind<(unsigned int)blolen;ind++) //raufintegriert
  {
    *(blo+ind) += *(blo+ind-1) + (char )inc;
  }
  for(ind=blolen-1;ind>0;ind--) //runterintegriert
  {
    *(blo+ind-1) += *(blo+ind) + (char) inc;
  }
  return true;
}
/*************************************************************************/
```

This procedure is fairly simple, predictable to the processor and fast. It performs mathematical integration of the block. It accepts two parameters "offs" and "inc". offs is added to the first byte prior to processing and inc is added to each byte. After reaching the block's end, the direction is reversed and integration is continued from the blocks end back to the start. It performs what is called diffusion in Shannon's terminology if fed with normal (i.e. not maliciously crafted) blocks.

Similar blocks may remain similar, however, if specially crafted for this purpose by an attacker. Thus bytinteg is not a cryptographic primitive per se and its cryptographic value - e.g. if used unprotectedly in a cipher as first step- would lie in adding restrictions to chosen plaintext attacks(so as to avoid initial diffusion). Key whitening of the plaintext block in a cipher completely removes this partial vulnerability and ensures efficient diffusion on using "bytinteg".

b) Procedure “bytadd_a“

The respective routine in C++ is listed below. It is contained in the Academic Signature module “helpersxx.cpp“.

```
/*****************************************************************************/
bool bytadd_a(int blolen, char* bladwell,  char* pertxt, int rounds, int spice, char fr)
//spice is individualizer(def 1 set in header),fr is flea loop multiplier(def 3 set in header)
{
  unsigned int ind,oi,k,hind,magic=0;
  unsigned char *blad;

  if(blolen<2)return false;
  if(fr<0||fr>10)fr=3; //default is 3 anyways
  blad=(unsigned char*)malloc(blolen);
  memcpy(blad,bladwell,blolen);
  for(k=0;k<(unsigned int)rounds;k++)
  {
      for(oi=0;oi<(unsigned int)blolen;oi++) //systematically working up
      {
          hind = (unsigned int)   *(blad +(( *(blad + oi))%blolen));
          hind+=(unsigned int)spice; //allow for differently spiced variants
          hind *= *(blad+((oi+1)%blolen)); // determine target index
          hind %= blolen;
          if(hind!=oi) *(blad + oi) += rotbyte(*(blad+hind),((unsigned char)(hind+oi)&7));
          *(blad+oi)+=(unsigned char)spice+(unsigned char)oi;
          if(!(oi&2)) *(blad+hind) = *(blad+hind) ^255; //entropizer step
      }
      for(oi=(unsigned int)blolen-1;oi>0;oi--) //systematically working down
      {
          hind = (unsigned int)   *(blad +(( *(blad + oi))%blolen));
          hind+=(unsigned int)spice; //allow for differently spiced variants
          hind *= *(blad+((oi+1)%blolen)); // determine target index
          hind %= blolen;
          if(hind!=oi) *(blad + oi) += rotbyte(*(blad+hind),((unsigned char)(hind+oi)&7));
          *(blad+oi)+=(unsigned char)spice+(unsigned char)oi;
          if(!(oi&2)) *(blad+hind) = *(blad+hind) ^255; //entropizer step
      }
      ind=blolen/2;
      for(oi=0;oi<fr*(unsigned int)blolen;oi++) //second loop statistically jumping
      {
          hind = (unsigned int)*(blad+ind)%blolen; // determine target index
          hind +=magic;
          magic += *(blad + oi%blolen) +(unsigned int)spice;
          hind %= blolen;
          if(hind!= ind) *(blad + ind) += rotbyte(*(blad+hind),(unsigned char)(ind+hind)&7);
          *(blad+ind)+=spice+(unsigned char)oi;
          if(oi&2) *(blad+hind) = *(blad+hind) ^255; //entropizer
          ind = (ind +hind+oi+spice)%blolen; //reset entry index
     }
  }
  memcpy(pertxt,blad,blolen);
  if(blad != NULL) free( blad);
  return true;
}
/***********************************/
```

This routine is one of the working horses of the Fleas algorithms. It contains three for loops. The first two loops work on the bytes of the block systematically. Loop one does it ascending, from byte 0 to the highest byte. Loop 2 does it descendingly working down systematically again. The third loop works on the bytes in an erratic pattern which is determined self referentially. A basic code element is contained in all three for loops.

Starting from low or high byte:

a) select the byte of the block, the target byte is pointing to.

b) add the spice parameter to the byte.

c) multiply with the following byte in the block

d) modulo reduce with respect to the block length to obtain the final reference to another

byte of the block.

e) if the byte pointed to is not the target byte itself, take the value of the byte pointed to and cyclically rotate it by the sum of reference index and loop index and add this value to the target byte.

f) Add the spice byte and the low byte of the loop index to the target byte.

g) if bit 1 of the loop index is 0 invert all bit values of the reference byte.

In the third loop, the bytes are not worked through systematically but by jumping erratically from byte to byte up to a multiple of block size specified in the parameter "fr". Additionally the core loop content is augmented somewhat.

Do for "fr" * block length times:

a) Select the target byte of the block as pointed to by variable "ind" (start value of ind is half of the block length).

b) Take the value of the target byte and add variable "magic" to this value (startvalue of magic is 0).

c) update "magic" by adding spice and the current loop index to it.

d) modulo reduce the value from b) with respect to the block length to obtain the final reference to a byte of the block.

e) if the byte pointed to is not the target byte itself, take the value of the byte pointed to and cyclically rotate it by the sum of reference index and the value of "ind" (=target index) and add this value to the target byte.

f) Add the spice byte and the low byte of the loop index to the target byte.

g) if bit 1 of the loop index is 0 invert all bit values of the reference byte.

h) select the next target index "ind" by adding to the previous "ind" the recent reference index, the loop index and the value of spice and modulo reduce with respect to block length.

That's it.

I admit that the verbal description is hard to follow, but the procedure was not primarily designed for good descriptiveness. It defies attempts to document it traditionally in nice block diagrams. The code itself is compact though. If you are used to reading C++ code, it might be best to just look at the code as presented above instead of following through the verbal description.

Since the procedure is rich in conditional execution of code and highly self referential regarding the block worked on, there is no static pattern of diffusion and/or confusion as in traditional ciphers or Pseudo Random Functions. The input block itself determines the mixing pattern. The unpredictable memory access pattern leads to permanent frustration of the processors attempts to employ pipelining and prefetching and thus tends to leave these speed enhancements of modern processors unused.

The procedure per se may be vulnerable to backtracing and thus may need proper precautions to prevent that to achieve preimage resistance. For this reason it should always be used in a multipath parallel arrangement sealed with a final uniting Xor like the one shown in the calling procedure "develop_ln2". In Academic Signature this procedure together with "bytmult_a" can be used in a 2,3 or 4 path pattern as round function in a Feistel network. The respective ciphers are called Fleas_lb(2), Fleas_lc(3) and Fleas_ld(4).

As mentioned above, the input block itself determines the mixing pattern. Thus, if used in a cipher, there may be a higher vulnerability at the input side towards chosen plaintext attacks than with traditional ciphers. Thus it is advisable to always employ some form of key whitening prior to supplying blocks to this procedure. This is consistently done in Academic Signature.

c) Procedure "bytmult_a"

The respective routine in C++ is listed below. It is contained in the Academic Signature module "helpersxx.cpp".

```
/************************************************************/
bool bytmult_a(int blolen, char* bladwell, char* pertxt, int rounds, int spice,char fr)
{
  unsigned int ind,hind,ihl,oi,magic,sl;
  int k;
  unsigned char *blad,*zsto;

  if(blolen<2)return false;
  if(fr<0||fr>10)fr=3; //default is 3 anyways

  blad=(unsigned char*)malloc(blolen);
  memcpy(blad,bladwell,blolen);
  magic=(unsigned int) (blolen/2);
  for(k=0;k<rounds;k++)
  {
      for(oi=0;oi<(unsigned int)blolen;oi++) //first loop systematically up
      {
                // determine pointed-to pointed to index
          magic += (ulong32)   *(blad + (*(blad+oi)%blolen));
          hind = (oi + magic+spice)%blolen; //add with oi for homogeneous index migration
          if(magic&4) { *(blad+oi) ^= 255;} //invert every now and then
          ihl=oi+((unsigned int) *(blad+hind)) * ((unsigned int) *(blad+oi%blolen));
          ihl += (oi + spice +magic)%257;
         *(blad+oi)^=  ((unsigned char)ihl);
          if(hind&1) { *(blad+oi) ^= 255;} //entropy enhancer
          *(blad+hind)= rotbyte(*(blad+hind),((unsigned char)(oi))&7);  //rotator
      }
      for(oi=(unsigned int)blolen-1;oi>0;oi--) //second loop systematically down
      {
                // determine pointed-to pointed to index
          magic += (ulong32)   *(blad + (*(blad+oi)%blolen));
          hind = (oi + magic+spice)%blolen; //add with oi for homogeneous index migration
          if(magic&4) { *(blad+oi) ^= 255;} //invert every now and then
          ihl=oi+((unsigned int) *(blad+hind)) * ((unsigned int) *(blad+oi%blolen));
          //hier noch ein entropizer für ihl rein
          ihl += (oi + spice +magic)%257;
         *(blad+oi)^=  ((unsigned char)ihl); //+ (unsigned char)(ihl>>8); //press into one byte
          if(hind&1) { *(blad+oi) ^= 255;} //entropy enhancer
          *(blad+hind)= rotbyte(*(blad+hind),((unsigned char)(oi))&7);  //rotator
      }
      ind = blolen/2; //initially set to middle of block
      zsto=(unsigned char *) malloc(blolen); //allocate space vor intermediate storage
      memcpy(zsto,blad,blolen);//fill with result of first loop
      for(oi=0;oi<(unsigned int)blolen*fr;oi++) //second loop, jump self referentially
      {
          magic += *(blad + *(blad+((oi+ind)%blolen)) %blolen) ;
          hind = (oi + magic+spice)%blolen; //add with oi for homogeneous index migration
          if(magic&2) { *(blad+(oi%blolen)) ^= 255;} //invert passive block every now and then
          ihl=oi +(((unsigned int) *(blad+hind))) * ((unsigned int) *(blad+ind));
          //hier noch ein entropizer fuer ihl rein
          ihl += (oi + spice +magic)%257;
          *(zsto+ind)^=  ((unsigned char)ihl);
          if(oi&1) { *(zsto+ind) ^= 255;} //entropy enhancer
          *(zsto+ind)= rotbyte(*(zsto+ind),((unsigned char)(ind))&7); //rotator
          ind = hind; //use last index as new reference index
      }
      memcpy(blad,zsto,blolen); //overwrite first loop result with second loop result
      free(zsto); // free intermediate storage
  }
  memcpy(pertxt,blad,blolen);// copy result to output block
  if(blad != NULL) free( blad);
  return true;
}
```

```
/************************************************************/
```

This procedure contains three for loops. The first two loops work on the bytes of the block systematically. Loop one does it ascending from byte 0 to the highest byte, loop 2 does it descendingly working down systematically again. The third loop works on the bytes in an erratic pattern which is determined self referentially. A basic code element is contained in all three for loops.

Starting from low or high byte:

a) update the parameter "magic" by adding a double dereferenced byte from the block. (See the C++ code above.)

b) Determine the reference index "hind" by adding loop index, "magic" and "spice" and modulo reduce it with respect to block size.

c) If bit 4 of the variable magic is 1, invert all bit values of the target byte.

d) Determine another parameter "ihl" by adding loop index with the product of the byte referenced to by "hind" with the target byte.

e) Add to "ihl" the sum of loop index, "spice" and "magic" modulo reduced by 257.

f) Xor the target byte with the low byte of parameter "ihl".

g) if bit 1 of parameter "hind" is 1, invert all bit values of the target byte.

h) Cyclically rotate the byte pointed to by hind loop index times.

In the third loop, the bytes are not worked through systematically but by jumping erratically from byte to byte up to a multiple of block size specified in the parameter "fr". Additionally the core loop content is augmented somewhat. Start at index blocklength/2.

Copy the intermediate block to an an additional memory location.

Do for "fr" * block length times:

a) update the parameter "magic" by adding a double dereferenced byte from the intermediate block.(See the C++ code above.)

b) determine the reference index "hind" by adding loop index, "magic" and "spice" and modulo reduce it with respect to block size.

c) If bit 2 of the variable magic is 1, invert all bit values of the target byte in the intermediate block.

d) Determine another parameter "ihl" by adding loop index with the product of the byte referenced to by "hind" in the intermediate block with the target byte.

e) Add to "ihl" the sum of loop index, "spice" and "magic" modulo reduced by 257.

f) Xor the target byte in the final block with the low byte of parameter "ihl".

g) If bit 1 of the loop index is 1, invert all bits of the target byte in the final block.

h) Cyclically rotate the target byte in the final block by the amount given by the target index.

I) Use last reference index "hind" as new target index.

I admit that the verbal description is even harder to follow than for "bytadd_a". Again, the procedure was not primarily designed for good descriptiveness. It defies attempts to document it traditionally in nice block diagrams. The code itself is almost as compact as in "bytadd_a". If you are used to reading C++ code, it might be best to just look at the code as presented above instead of following through the verbal description.

Since this procedure is rich in conditional execution of code too and highly self referential

regarding the block worked on, there is no static pattern of diffusion and/or confusion as in traditional ciphers. The input block itself constitutes the mixing pattern. The unpredictable memory access pattern leads to permanent frustration of the processors attempts to employ pipelining and prefetching and thus tends to leave these speed enhancements of modern processors unused.

This procedure per se may also be vulnerable to backtracing and thus may need the same precautions as “bytadd_a“ to prevent that. It should also be used in a multipath parallel arrangement sealed with a final uniting Xor like the one shown in the calling procedure "develop_ln2". Let me repeat here that in Academic Signature this procedure together with "bytadd_a" can be used in a 2,3 or 4 path pattern as round function in a Feistel network. The respective ciphers are called Fleas_lb(2), Fleas_lc(3) and Fleas_ld(4).

The same argument as given for "bytadd_a" applies again regarding the necessity of key whitening if used in a cipher.